TPLP **21** (5): 543–556, 2021. © The Author(s), 2021. Published by Cambridge University Press
doi:10.1017/S1471068421000260 First published online 23 September 2021

Optimizing Probabilities in Probabilistic Logic Programs

DAMIANO AZZOLINI
Dipartimento di Ingegneria - University of Ferrara, Via Saragat 1, I-44122, Ferrara, Italy
(*e-mail:* damiano.azzolini@unife.it)

FABRIZIO RIGUZZI
Dipartimento di Matematica e Informatica - University of Ferrara, Via Saragat 1, I-44122, Ferrara, Italy
(*e-mail:* fabrizio.riguzzi@unife.it)

submitted 6 August 2021; accepted 22 August 2021

Abstract

Probabilistic logic programming is an effective formalism for encoding problems characterized by uncertainty. Some of these problems may require the optimization of probability values subject to constraints among probability distributions of random variables. Here, we introduce a new class of probabilistic logic programs, namely probabilistic *optimizable* logic programs, and we provide an effective algorithm to find the best assignment to probabilities of random variables, such that a set of constraints is satisfied and an objective function is optimized.

KEYWORDS: probabilistic logic programming, optimization, constraints

1 Introduction

Uncertainty on the data is ubiquitous and pervades almost all domains, such as social networks (uncertainty on the relations), power networks (uncertainty on the components), and road networks (uncertain about the traffic). Many real-world applications that can be modelled with random variables require the solution of optimization problems. In particular, considering the domains listed above, in a social network or in a collaboration network we may want to optimize the probability of targeting some people (or a group of them) during a marketing campaign, but we may be uncertain whether the people (or group of people) know each other, so we need to introduce random variables to model the data. In road networks we may want to minimize the probability of encountering a traffic jam, but typically we are unsure about the traffic distribution, so probabilities are needed. In power networks we can, for example, optimize the positioning of several electronic components to reduce the probability of a system fault, given that these components have a certain tolerance and a performance degradation after several running hours. Furthermore, uncertainty is often an intrinsic characteristic of the data itself (collected by faulty sources, incomplete, ...), so probabilities are necessary. The same type of task can as well be of interest for other, more theoretical, domains; just to name a few: in Markov network we can set the probability of the possible transitions to optimize

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

the chances of reaching a certain final state, in probabilistic context-free grammars we may be interested in finding the probability of the various components such that the classification of a sentence does not change.

Probabilistic logic programming (PLP) (De Raedt *et al.* 2008; Riguzzi 2018) is an extension of Logic Programming that allows to encode probabilistic models with a rich language. In this paper, we address the problem of optimizing an objective function by tuning the probabilities of probabilistic logic programs subject to constraints on probabilities of facts and queries.

To solve the aforementioned task, we extend the PITA reasoner (Riguzzi and Swift 2010) to allow the definition of *optimizable* facts, that is, facts whose probabilities can be set in order to optimize (minimize or maximize) an objective function subject to constraints. Constraints can be on both the query and the optimizable facts. We introduce the definition of Probabilistic Optimization Logic Program and Probabilistic Optimizable Problem, and provide an algorithm to solve it. This task can be considered as constrained parameter learning.

By extending the library PITA, we can express models that retain the full logic programs with annotated disjunction (LPAD) expressive power, while adding new optimizable facts: in this way, we do not need to rewrite a specialized system to solve the problem, we just need to build upon the previous developed work.

The paper is organized as follows: Section 2 analyzes the related works, Section 3 presents the basic definitions regarding PLP, and Section 4 introduces the definition of the task we consider, together with an algorithm to solve it. Section 5 presents some empirical results and Section 6 concludes the paper.

2 Related works

A huge part of related works focuses on parameter learning, that is, learning the parameters of a probabilistic logic program in a supervised manner, starting from a set of examples. However, explicit constraints on probabilities of probabilistic facts are usually not considered. Here, we learn the parameter (probabilities) of the program's facts starting from a set of constraints, so our algorithm does not need a training set, since it can be classified as an algorithm for solving constrained optimization problems.

One of the first approaches to solve the parameter learning task can be found in PRISM (Sato 1995): the goal is to find the maximum likelihood parameters of special atoms, called *msw* atoms, that are not present in the dataset. To learn the parameters, they propose a naive implementation of the EM algorithm, later improved by Sato and Kameya (2001). EM is also used by Bellodi and Riguzzi (2012) to learn the parameters of LPADs (Vennekens *et al.* 2004) represented as binary decision diagrams (BDDs). Another technique is presented in ProbLog2 (Fierens *et al.* 2015) that includes the algorithm LFIProbLog (Gutmann *et al.* 2011), where parameters of ProbLog programs are learned from partial interpretations.

Parameter learning using gradient-based methods is applied by Gutmann *et al.* (2008), where derivatives are computed directly on the BDD representation of the program. Here, we use gradient-based methods as well (see Section 5), but we do not compute gradients directly on the BDD. Rather, we obtain the equation represented by the BDD and then use this equation (after simplification) in all the calculations. In this way, we traverse the

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

BDD only once. A related idea is introduced by Kimmig *et al.* (2011) where the authors present aProbLog, an extension of the programming language ProbLog (De Raedt *et al.* 2007). It allows to solve different tasks, among them, the sensitivity analysis, that is, see how a change in the parameters affects the probability of a query. However, they cite the task in passing, and they do not consider further constraints on the probability of the variables. The same problem is discussed in a paper by Orsini *et al.* (2017). Other approaches, such as DTProblog (Van den Broeck *et al.* 2010), extend probabilistic logic programs with Boolean decision variables. The goal is to find the optimal set of these variables to maximize the overall expected utility. As happens for previously described approaches, no constraints are considered.

Another related solution is presented by Latour *et al.* (2017), where the authors introduce an algorithm to combine Probabilistic (Logic) Programming and Constraint Programming to solve decision-theoretic tasks: as in that paper, we use a compact representation of the probabilistic logic program (they use sentential decision diagram (SDD), we use BDD), and extend an already existing tool (they extend ProbLog, we extend PITA). Differently, they restrict the type of constraints involved (linear constraints over sum of Boolean decision variables) and they do not consider the computation of optimal probability values.

A research area that combines probability and constraints is stochastic constraint programming (SCP) (Walsh 2002). SCP programs are composed of two types of variables: decision variables that can be set, and stochastic variables that follow a probability distribution. The goal is to find a subset of decision variables such that constraints are satisfied with a certain probability. Here, we do not consider probabilistic constraints, we have hard constraints that must be always true, and we search for optimal probability values for optimizable facts that are present in the program (they cannot be removed, as happens for decision variables) but with a tunable probability. Furthermore, these approaches are often tailored to solve specific problems such as games (Antuori and Richoux 2019), scheduling (Lombardi and Milano 2010), or sequential planning (Babaki *et al.* 2017).

3 Probabilistic logic programming

Logic Programming offers the possibility to encode problems with a high level of abstraction, allowing programmers to focus on the overall structure, rather than on implementation details. However, logic programs are not well suited to handle uncertainty, an intrinsic feature of real-world scenarios. PLP (De Raedt *et al.* 2008; Riguzzi 2018) extends logic programming by adding probabilistic facts that allow to reason with probabilities. Following the ProbLog (De Raedt *et al.* 2007) syntax, a probabilistic fact f_i has the form

$$\Pi_i :: f_i,$$

where f_i is an atom and $\Pi_i \in]0, 1]$, with the meaning that each ground instantiation of f_i is true with probability Π_i and false with probability $1 - \Pi_i$. Probabilistic facts are considered independent: this may seem a severe restriction but, in practice, does not limit the expressiveness of the language (Riguzzi 2018). If we consider the following illustrative example:

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

```
0.5::no_traffic.
0.9::no_accidents.

on_time:- no_traffic, no_accidents.
```

we can ask the probability that a driver will arrive on time at work (`on_time`), which is $0.5 \cdot 0.9 = 0.45$. Similarly, an LPAD is composed of a finite set of probabilistic disjunctive clauses of the form

$$h_1 : \Pi_1; \ldots; h_n : \Pi_n : -b_1, \ldots, b_m.$$

where the semicolon stands for disjunction, $h_1, \ldots h_n$ are logical atoms and $b_1, \ldots, b_m$ are logical literals. $\Pi_1, \ldots, \Pi_n$ are real numbers in the interval $[0, 1]$ that sum to 1. If this is not true, the head of the annotated disjunctive clause implicitly contains an extra atom *null* that does not appear in the body of any clause; its annotation is $1 - \sum_{k=1}^{n} \Pi_k$.

One of the most followed semantics, the Distribution Semantics (Sato 1995), is at the foundations of several PLP systems. Here, we consider the Distribution Semantics for programs with a finite Herbrand base (without function symbols). An *atomic choice* is a selection or not of a grounding $f\theta$ of a probabilistic fact $\Pi :: f$. It is usually indicated with (f, θ, s), where $s \in \{0, 1\}$: if $s = 1$ the fact is selected, if $s = 0$ it is not. A set of atomic choices is *consistent* if only one of the two alternatives for a probabilistic fact is considered, that is, it does not contain two atomic choices (f, θ, s) and (f, θ, t) with $s \neq t$. A consistent set of atomic choices forms a *composite choice*, and if it contains one atomic choice for every grounding of each probabilistic fact it is called *selection*. The probability of a composite choice c can be computed as:

$$P(c) = \prod_{(f_i, \theta, 1) \in c} \Pi_i \cdot \prod_{(f_i, \theta, 0) \in c} (1 - \Pi_i).$$

A selection identifies a *world* (a Logic Program) and its probability can be computed as the product of the probabilities of the atomic choices. The programs do not contain function symbols, thus the set of worlds is finite and the probabilities of all the worlds add up to 1. Given a ground query q, the conditional probability of q given a world w, represented with $P(q \mid w)$, is 1 if the query is true in the world, 0 otherwise. So, the probability of a (ground) query q can be obtained marginalizing the joint probability distribution between the worlds w and the query, $P(q, w)$. In formulas:

$$P(q) = \sum_{w} P(q, w) = \sum_{w} P(q \mid w) \cdot P(w) = \sum_{w \models q} P(w).$$

A composite choice is an *explanation* for a query q (a conjunction of ground atoms) if q is true in every world identified by the composite choice. If every world in which q is true belongs to the worlds identified by a set of explanations, the set is named *covering*. The probability of a query can also be defined as the probability measure of a covering set of explanations.

A covering set of explanations for a query can be represented as a Boolean formula in disjunctive normal form (DNF). To perform inference on these formulas, usually probabilistic logic programs are compiled into a target language using knowledge compilation (Darwiche and Marquis 2002). Among all the possibilities, BDDs and SDDs are the most used, since they offer a compact representation. Between the two, SDDs are more

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

compact than BDDs. However, BDDs are managed by optimized libraries and usually offer comparable performance. In this paper, we consider BDDs.

A BDD is a direct acyclic graph where each node (that in the PLP setting represents a probabilistic fact) has two edges: the 0 (false) edge and the 1 (true) edge. There are only two types of terminal nodes: 0 (false) and 1 (true). Some packages introduce the possibility to have a third edge, the 0-complemented edge, with the meaning that the function represented by the subtree must be complemented. Here we use this convention. With this third type, the 0-terminal is not needed. Usually, the 0-edge is graphically represented with a dashed line, the 1-edge with a solid line and the 0-complemented edge with a dotted line (see Figure 2 with the running example used in the paper that will be discussed later).

4 Probabilistic optimizable logic programs

We denote *optimizable* facts with the special functor `optimizable`. They have the following syntax:

$$\text{optimizable } [\Pi_{lb}, \Pi_{ub}] :: a.$$

where a is a logical term, and Π_{lb} and Π_{ub} are the lower and upper probabilities for the term ($\Pi_{lb} < \Pi_{ub}$). In our implementation, if lower and upper probabilities are not specified, the range is by default $[0.001, 0.999]$. Intuitively, these are the facts whose probabilities can be changed to minimize an objective function subject to constraints. Both the objective function (that may not be the same of the query) and the constraints can be linear or nonlinear combinations of optimizable facts. As happens for probabilistic facts, also optimizable facts are considered independent. We now introduce the definition of *probabilistic optimizable logic program* (POLP) and *probabilistic optimizable problem.*

Definition 1 (*Probabilistic Optimizable Logic Program*)
Given an LPAD $\mathcal{L}$, a set of optimizable facts $\mathcal{O}$, an objective function $\mathcal{F}$, and a set of constraints $\mathcal{C}$, the tuple $(\mathcal{L}, \mathcal{O}, \mathcal{F}, \mathcal{C})$ identifies a POLP.

Definition 2 (*Probabilistic Optimizable Problem*)
Given a POLP $(\mathcal{L}, \mathcal{O}, \mathcal{F}, \mathcal{C})$, and a conjunction of ground atoms, the *query* (q), the probabilistic optimizable problem consists of two steps:

- find a probability assignment $\mathcal{A}^*$ to optimizable facts $o_i \in \mathcal{O}$ such that the objective function is minimized (or maximized) and constraints are not violated, that is:

$$\mathcal{A}^* = \arg\min_{\mathcal{A}}(\mathcal{F} \mid \mathcal{C}, \mathcal{A}),$$

- compute the probability of the query given these assignments

$$P(q \mid \mathcal{A}^*),$$

This task can also be considered as parameter learning under constraints.

Consider the following motivating example:

Example 1
Suppose you need to route a signal through a path composed of intermittent edges (connections), from a source to a given destination. Some of the edges have a fixed

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

```
0.9::edge(a,b).
optimizable [0.3,0.8]::edge(b,c).
optimizable [0.3,0.8]::edge(b,d).
0.3::edge(c,e).
0.8::edge(d,e).

path(X,X).
path(X,Y):- path(X,Z), edge(Z,Y).
```

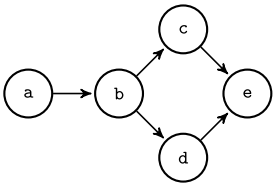


Fig. 1. Program for motivating example (left), together with the represented graph (right).

probability to be active, while some other edges can be controlled, and their probabilities can be set. The constraint is that the probability to reach the destination must be above a certain threshold. The objective function is that the probability of the edges that can be controlled should be set at the minimal value to reach the target probability, since setting higher probabilities also requires higher manufacturing costs. Moreover, the probabilities of the edges should be similar (i.e. as close as possible).

Let us model the previous scenario with a POLP. Consider five nodes named `a`, `b`, `c`, `d` and `e`. Their connections can be represented with `edge/2` facts. Suppose that edges between `a` and `b`, `c` and `e`, and `d` and `e` cannot be controlled and have, respectively, probability 0.9, 0.3 and 0.8. Edges between `b` and `c` and `b` and `d` can be controlled and must have probability in the range $[0.3, 0.8]$. A path between two nodes is represented by predicate `path/2` (see Figure 1). The goal is to minimize the sum of the probabilities of optimizable variables (`edge(b,c)` and `edge(b,d)`) given that the probability to reach `e` from `a` (`path(a,e)`) must be above a certain threshold. Constraints also impose that the difference between the probabilities of the two optimizable facts should be less than, for example, 0.1. The POLP shown in Figure 1 represents the described situation.

To simplify the notation, in the remaining part of the paper we remove the explicit probability signature from facts involved in the optimization task. For example, $edge(b, c)$ represents the probability of $edge(b, c)$ ($P(edge(b, c))$). If the goal is to minimize the sum of the probabilities of `edge(b,c)` and `edge(b,d)`, the lower bound probability for query `path(a,d)` is 0.6, and the difference between the probabilities of the optimizable facts should be less than 0.1, following Def. 1 we get:

- $\mathcal{O} = \{edge(b, c), edge(b, d)\}$
- $\mathcal{F} = minimize(edge(b, d) + edge(b, c))$
- $\mathcal{C} = \{path(a, d) > 0.6, edge(b, d) \in [0.3, 0.8], edge(b, c) \in [0.3, 0.8],$
  $edge(b, c) - edge(b, d) < 0.1, edge(b, d) - edge(b, c) < 0.1\}$

Implicitly, $path(a, d) \in [0, 1]$.

To solve the probabilistic optimizable problem, we introduce the special predicate

```
prob_optimize/4
```

that receives as input the query (in our example `path(a,e)`), the objective function to be minimized (`edge(b,c) + edge(b,d)`), and a list of constraints. As output, it returns the probability of the query and the optimal probability assignment to optimizable facts. So, for the previous example, the query would be

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

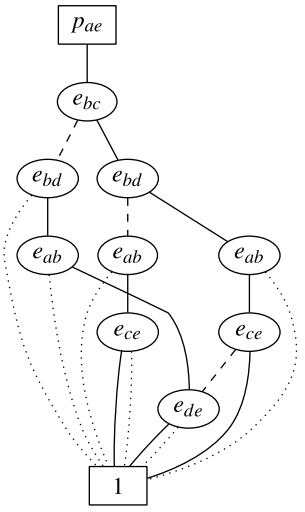


Fig. 2. BDD for the program shown in Figure 1, where a dashed line represents a 0-edge, a solid line the 1-edge, and a dotted line the 0-complemented edge.

```
prob_optimize(
 path(a,e),
 [edge(b,c) + edge(b,d)],
 [path(a,e) > 0.6, edge(b,c) - edge(b,d) < 0.1,
 edge(b,d) - edge(b,c) < 0.1],
 Assignments).
```

Lower and upper bounds are directly considered by the predicate, without the need to be specified also in the constraints list.

The main idea is that a POLP induces a (non) linear function $f : \mathbb{R}^n \to \mathbb{R}$, where n is equal to the number of optimizable facts. In the program shown in Figure 1, $n = 2$. Only if $n = 1$ the function is linear since there is only one variable involved (with $n > 1$ we have a joint probability distribution, so an equation with several variables multiplied together). Otherwise, there are several variables multiplied together. Consider Figure 2 that represents the BDD for the query `path(a,e)`, denoted with p_{ae}, where e_{xy} stands for `edge(x,y)`. There are three paths with probability greater than 0. The induced function is the sum of the functions induced by the three paths. For example, the leftmost path that goes to terminal 1 with a regular arc induces the following equation: $f(e_{bc}, e_{bd}) = (1 - e_{bc}) \cdot e_{bd} \cdot 0.72$, where $0.72 = e_{ab} \cdot e_{de}$.

To solve the optimization problem, the following operations are performed: first, the BDD for the query is built, then paths and their probabilities are computed using Algorithm 2. It goes as follows: first, the BDD is reordered to have the optimizable variables first. Then, the function PathsProbRec is recursively called until a node associated with a not-optimizable variable is found (including terminal nodes). From there, to compute probabilities, function Prob presented by De Raedt *et al.* (2007) is called and an empty path list and the computed probability are returned. Essentially, function Prob recursively analyzes the BDD until the terminal is found, and, from there, probabilities are computed with a dynamic programming algorithm. It is reported in Algorithm 3 for

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

Algorithm 1 Function OPTIMIZEPROB: optimization of probability of random variables.

```
1: function OptimizeProb(query,objective,constraintsList,algorithm)
2:     root ← Compute the BDD for the query
3:     paths ← PathsProb(root)
4:     query equation ← convert paths into symbolic equation
5:     Simplify query equation
6:     Replace constraints from constraintsList involving the query with query equation
7:     assignments ← Call the non-linear optimization solver with [objective, constraintsList, algorithm]
8:     prob ← evaluate(query equation | assignments)
9:     return [assignments, prob]
10: end function
```

clarity. When the node is associated with an optimizable fact, the paths computed at the true and false child are extended with the current node index, provided that the path has probability greater than 0. To speed up computation, already computed paths as well as already computed probabilities (used in function PROB), are stored in a table: in this way, when an already encountered node is found, the paths and the probabilities are retrieved from the memory, rather than recomputed.

From the list of paths, where optimizable variables are kept symbolically, we obtain an equation, that we call it *query equation* in this way: we multiply together all the nodes n_i of the paths and their probability for each path in the list of paths (if the node is selected as 0, we multiply by $1 - n_i$) and add all the results. So, for example, for Figure 2, the resulting list of paths obtained by Algorithm 2 is $[[0.774, [[e_{bd}, 1], [e_{bc}, 1]]], [0.27, [[e_{bd}, 0], [e_{bc}, 1]]], [0.72, [[e_{bd}, 1], [e_{bc}, 0]]]]$, where $0.774 = e_{ab} \cdot (e_{ce} + (1 - e_{ce}) \cdot e_{de})$, $0.72 = e_{ab} \cdot e_{de}$, and $0.27 = e_{ab} \cdot e_{ce}$ (obtained by following the correspondent paths on the BDD), which represents the equation $e_{bd} \cdot e_{bc} \cdot 0.774 + (1 - e_{bd}) \cdot e_{bc} \cdot 0.27 + e_{bd} \cdot (1 - e_{bc}) \cdot 0.72$. The symbolic equation is then simplified and substituted in the constraint(s) involving the query. This equation, along with all constraints specified by the user, are added to the (nonlinear) optimization problem and passed to a nonlinear optimization solver. Finally, once the optimal value(s) are calculated, the probability of the query is computed by evaluating the query equation, where the symbolical optimizable variables are substituted by their optimal values. The whole process is reported in Algorithm 1. If we consider the program shown in Figure 1, one of the possible solutions is given by $f(0.6352, 0.7352) = 1.3704$, which yields a probability of the query of 0.6.

The complexity of answering probabilistic queries is, in general, #P-complete (Koller and Friedman 2009). The construction of the BDD starting from the derivations of the program is as well #P-complete, and so the solution of the Probabilistic Optimizable Problem is, at least, in that class. Finding the optimal order of the variables that minimizes the size of a BDD (and thus reduces the size of the symbolic equation) is a NP-complete task (Bollig and Wegener 1996), so it is generally infeasible. Moreover, if we choose this approach, every time we reorder the BDD we need to traverse it again to extract a new equation, and check whether it is more compact than the previously obtained. Even if the traversal of a BDD can be performed polynomially (we visit every node only once), the number of possible combinations of variables is too large to check exhaustively. For these reasons, we decided to extract the equation directly from the BDD after having reordered it by moving nodes that correspond to optimizable facts first. The reorder can be performed polynomially in the size of the BDD by swapping adjacent variables (Jiang *et al.* 2017).

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

The reordering of the BDD is crucial since it allows to directly use function PROB and obtain a more compact symbolic representation, where the combinations of the optimizable variables are multiplied by a single numerical value. Similarly, simplification of the query equation is fundamental, since it greatly reduces its size, resulting in faster computation. Consider, for example, the equation represented by the BDD shown in Figure 2 written before and reported here for clarity: $e_{bd}\cdot e_{bc}\cdot 0.774+(1-e_{bd})\cdot e_{bc}\cdot 0.27+e_{bd}\cdot(1-e_{bc})\cdot 0.72$. After the simplification, it is reduced as: $-0.216\cdot e_{bc}\cdot e_{bd}+0.27\cdot e_{bc}+0.72\cdot e_{bd}$ (the number of operations is now 6, previously was 10). Moreover, this process may also reduce the impact of the BDD structure, since the order of the variables in the BDD also determines its size: several BDDs can represent the same function, but some of these BDDs may not be optimal in terms of compactness. If we directly utilize the query equation extracted by Algorithm 2, without simplifying the result, we may obtain a long equation that involves more computations (multiplications and summations) than the strict necessary. Since the query equation can be called several times during the solution of the optimization problem, not simplifying it can largely increase the overall execution time.

5 Experiments

We implemented the algorithm presented before[1] using C and exploiting the library for nonlinear optimization NLopt (Johnson 2020) to solve the optimization problem and the library CUDD (Somenzi 2015) for the operations on BDD. The Probabilistic Logic Part is managed by SWI-Prolog (Wielemaker *et al*. 2012) version 8.3.15. The simplification of the query equation is performed with the function `simplify` from Python SymPy package (Meurer *et al*. 2017): guided by some heuristics, it iteratively applies some simplifications to the equation to reduce its size. However, in general, there is no guarantee that the minimal size is found.

To test our algorithm on real-world scenarios described in Section 1, we selected from Rossi and Ahmed (2015) several dataset characterized by different structures, among them: social network, collaboration network, road connections, and power network. We pre-process the data by converting the matrix in a set of facts of the form `edge(a,b)` (eventually prepending the functor `optimizable` and adding a probability range or value) to represent a connection between nodes `a` and `b`. We choose these datasets since they represent several real-world networks, linked to the motivations listed in Section 1. We selected to test the implementation on, among other, social networks, collaboration networks, road networks, and power networks. For all of these, the goal is to constrain the probability of the paths between a random source (`Source`) and a random destination (`Dest`), provided that the path between these two nodes exists, to be greater than 0.8, while minimizing the sum of the probabilities of optimizable edges. Note that we minimized the sum of the probabilities of all the optimizable facts, not only the ones involved in the query, since it may be difficult to spot which facts are involved in it. For example, if you want to compute all the paths from a source to a destination, it is difficult to say which edges will be involved, without running the program. The number of optimizable

[1] Source code available at: https://bitbucket.org/machinelearningunife/polp_experiments

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

Algorithm 2 Function PathsProb: computation of all the paths of a BDD and of their probability.

```
1: function PathsProb(root)
2:     root' ← Reorder(root)                                   ▷ BDD reordering
3:     TablePaths ← ∅
4:     TableProb ← ∅
5:     if root'.comp then
6:         comp ← true
7:     else
8:         comp ← false
9:     end if
10:    return PathsProbRec(root', comp)
11: end function
12: function PathsProbRec(node, comp)
13:    comp ← node.comp ⊕ comp
14:    if var(node) is not associated to an optimizable fact then
15:        p ←Prob(node)                                       ▷ Call to prob
16:        if comp then
17:            Res ← [1 − p, []]
18:        else
19:            Res ← [p, []]
20:        end if
21:    else
22:        if TablePaths(node.index) ≠ ∅ then
23:            return TablePaths(node.index)
24:        else
25:            Lp0 ← PathsProbRec(child0(node), comp)
26:            Lp1 ← PathsProbRec(child1(node), comp)
27:            Res ← []
28:            for all path ∈ Lp0 do
29:                if path.prob > 0 then
30:                    Res ← Res ∪ {path ∪ [node.index, 0]}
31:                end if
32:            end for
33:            for all path ∈ Lp1 do
34:                if path.prob > 0 then
35:                    Res ← Res ∪ {path ∪ [node.index, 1]}
36:                end if
37:            end for
38:        end if
39:        Add node.index → Res to TablePaths
40:    end if
41:    return Res
42: end function
```

Algorithm 3 Function Prob: computation of the probability of a BDD.

```
1: function Prob(node)
2:     if node is a terminal then
3:         return 1
4:     else
5:         if TableProb(node.pointer) ≠ null then
6:             return TableProb(node)
7:         else
8:             p0 ←Prob(child0(node))
9:             p1 ←Prob(child1(node))
10:            if child0(node).comp then
11:                p0 ← (1 − p0)
12:            end if
13:            Let π be the probability of being true of var(node)
14:            Res ← p1 · π + p0 · (1 − π)
15:            Add node.pointer → Res to TableProb
16:            return Res
17:        end if
18:    end if
19: end function
```

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

Table 1. *Results for networks experiments. C, M and S stand respectively for CCSAQ, MMA, and SLSQP algorithms.* `|V|` *and* `|E|` *are number of vertices and edges.*

	Features		Time (s)			Probability			StdDev (Prob)		
Dataset	`|V|`	`|E|`	C	M	S	C	M	S	C	M	S
bio	658	1129	2595	4934	**1072**	0.850	0.823	**0.800**	0.065	0.059	**0.004**
ca	379	914	2859	2137	**387**	0.821	0.823	**0.816**	**0.023**	0.031	0.030
DD244	291	882	2070	2355	**521**	0.822	0.815	**0.800**	0.037	0.030	**0.000**
E60	10	36	**25**	30	37	**0.887**	**0.887**	**0.887**	0.068	**0.065**	**0.065**
IIP	219	613	1079	690	**209**	0.802	**0.755**	0.801	**0.004**	0.190	**0.004**
p494	494	586	3026	3898	**1052**	0.832	0.820	**0.800**	0.041	0.029	**0.000**
p662	662	906	16,167	7990	**2292**	0.802	0.824	**0.800**	0.003	0.020	**0.000**
rtf	283	418	271	211	**48**	0.817	0.826	**0.800**	0.029	0.026	**0.000**
rc	39	170	47	35	**7**	0.837	0.864	**0.829**	0.063	0.065	**0.060**
rt	97	117	21	13	**3**	0.827	0.840	**0.802**	0.058	0.056	**0.004**
soc	16	58	133	133	**11**	**0.842**	**0.842**	**0.842**	**0.057**	**0.057**	**0.057**
web	265	530	829	712	**146**	0.821	0.821	**0.801**	0.057	0.044	**0.003**

facts (edges) is randomly set to 50% of the total number of edges, and the values lower and upper bounds of the probability of optimizable facts are respectively 0.001 and 0.999. The remaining facts are probabilistic facts with probability 0.5. Results are averages of 10 runs with randomly chosen source and destination. Facts (both optimizable and probabilistic) are of the form `edge/2`, and the query is `path(Source,Dest) > 0.8`, where `path/2` is the predicate described above (Figure 1).

In a second set of experiments, we generate complete graphs of increasing size. As before, the goal is to minimize the sum of the optimizable facts while the probability of the path between 1 and N (where N is the size of the graph) must be above 0.8. The subdivision between optimizable and probabilistic facts is the same as before (50–50) as well as the structure of the facts and the query (`edge/2` facts, `path(1,N) > 0.8`). For this experiment, the graph size (and thus the number of optimizable facts) is notably smaller than the previous dataset. However, the solution of the optimization problem is harder, since the graph is fully connected, so there are many paths from the root of the BDD to the terminal, resulting in a very long and complex constraint function.

For all the experiments, we tested three local gradient-based optimization algorithms available in NLopt (Johnson 2020): two based on conservative convex separable approximations (Svanberg 2002), denoted with MMA and CCSAQ, and one based on sequential quadratic programming (Kraft 1994), denoted with SLSQP.

Experiments were conducted on a cluster[2] with Intel® Xeon® E5-2630v3 running at 2.40 GHz. We set the tolerance to 10^{-5}, that is, the optimization process stops when the variation of the objective function is less than this value. The available memory for the execution has been set to 8GB. Execution times are computed with the SWI-Prolog predicate `statistics/2` with the keyword `runtime`. The maximum execution time for each experiment was 8 hours.

Results are shown in Tables 1 and 2, which report the average overall execution time (BDD generation, simplification and optimization) and the average probability for all three algorithms. For the first set of experiments, we also tabled the standard deviations

[2] https://www.fe.infn.it/coka/doku.php

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

Table 2. *Results for complete graphs experiments.*

N	C (s)	M (s)	S (s)	C (prob)	M (prob)	S (prob)
3	1.6	1.7	**0.4**	0.800	0.800	0.800
4	8.2	2.7	**0.4**	0.818	0.800	0.800
5	4.5	4.7	**0.9**	0.800	0.800	0.800
6	10.7	12.4	**1.3**	0.800	0.800	0.800
7	202.4	193.8	**30.1**	0.800	0.800	0.800
8	1360.3	1,600.6	**177.6**	0.800	0.800	0.800

of the 10 averaged probabilities and the number of vertices and edges for each dataset. We marked in bold the best algorithm for every experiment. In Table 1, the dataset bio stands for bio-DM-LC, ca for ca-netscience, E60 for ENZYMES_g60, IIP for internet-industry-partnerships, p949 for power-494-bus, p669 for power-662-bus, rtf for reptilia-tortoise-fi-2008, rc for road-chesapeake, rt for rt-retweet, soc for soc-tribes, and web for webkb-wisc (all available from Rossi and Ahmed (2015)).

The time for the BDD construction and for the query equation extraction is an order of magnitude less than the time used for the optimization, often negligible. This is also the case for the simplification of the equation (even if it is a little more expensive than the BDD extraction, and sometimes, for larger graphs, it requires several seconds), so we decided to not separate the three values. Rather, we provide only the sum of the execution times of the three operations.

Usually, the best algorithm (fastest and more accurate in terms of difference with the lower probability, 0.8 for these experiments) is SLSQP, with an important difference in terms of both execution time and probability in most of the cases. Especially, for the complete graph dataset, SLSQP requires almost ten times less the time required by CCSAQ and MMA for bigger graphs.

The execution time exceeds eight hours for complete graph of size 9. The main reason is that nodes have high degree and so the number of paths increases exponentially, resulting in an explosion of the length of the query equation. To see this phenomenon, consider the dataset bio-DM-LC and power-494-bus: the former has approximately 200 vertices and edges more than the latter. However, execution times are comparable, especially for SLSQP. This also occurs for different networks we tested, still taken from Rossi and Ahmed (2015), where some of the queries does not terminate in the time limit, such as soc-firm-hi-tech, ia-crime-moreno, lp_adlittle, and soc-wiki-vote.

The bottleneck of our approach is, as expected, in the solution of the optimization problem, rather than the construction of the BDD representing the program or the simplification of the query equation. One possible trivial solution to this problem is reducing the tolerance of the optimization algorithm, but at the cost of less accurate results. Alternatively, we can try to provide more compact representations of the programs (some alternatives to BDDs exist, as discussed at the beginning of the paper), or leverage techniques from lifted inference. These are all directions of possible interesting future works.

6 Conclusions

In this paper, we introduced the definition of POLP and proposed a simple yet effective algorithm to find the optimal probabilities of optimizable facts subject to constraints.

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

Empirical results show that the proposed solution is scalable also for datasets with several hundreds of vertices. As future work, we planned to test other optimization packages and apply this approach to programs with continuous random variables (Azzolini *et al*. 2019; 2021).

Competing interests The authors declare none.

References

Antuori, V. and Richoux, F. 2019. Constrained optimization under uncertainty for decision-making problems: Application to real-time strategy games. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, 458–465.

Azzolini, D., Riguzzi, F. and Lamma, E. 2021. A semantics for hybrid probabilistic logic programs with function symbols. *Artificial Intelligence 294*, 103452.

Azzolini, D., Riguzzi, F., Lamma, E. and Masotti, F. 2019. A comparison of MCMC sampling for probabilistic logic programming. In *Proceedings of the 18th Conference of the Italian Association for Artificial Intelligence (AI*IA 2019), Rende, Italy 19-22 November 2019*, M. Alviano, G. Greco, and F. Scarcello, Eds. Lecture Notes in Computer Science. Springer, Heidelberg, Germany.

Babaki, B., Guns, T. and de Raedt, L. 2017. Stochastic constraint programming with and-or branch-and-bound. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-2017*, 539–545.

Bellodi, E. and Riguzzi, F. 2012. Learning the structure of probabilistic logic programs. In *22nd International Conference on Inductive Logic Programming*, S. Muggleton, A. Tamaddoni-Nezhad and F. Lisi, Eds. Lecture Notes in Computer Science, vol. 7207. Springer Berlin Heidelberg, 61–75.

Bollig, B. and Wegener, I. 1996. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transaction on Computers 45,* 9, 993–1002.

Darwiche, A. and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research 17*, 229–264.

De Raedt, L., Frasconi, P., Kersting, K. and Muggleton, S., Eds. 2008. *Probabilistic Inductive Logic Programming*. LNCS, vol. 4911. Springer.

De Raedt, L., Kimmig, A. and Toivonen, H. 2007. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, M. M. Veloso, Ed., 2462–2467.

Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D. S., Gutmann, B., Thon, I., Janssens, G. and De Raedt, L. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming 15,* 3, 358–401.

Gutmann, B., Kimmig, A., Kersting, K. and De Raedt, L. 2008. Parameter learning in probabilistic databases: A least squares approach. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2008)*. Lecture Notes in Computer Science, vol. 5211. Springer, 473–488.

Gutmann, B., Thon, I. and De Raedt, L. 2011. Learning the parameters of probabilistic logic programs from interpretations. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2011)*, D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, Eds. Lecture Notes in Computer Science, vol. 6911. Springer, 581–596.

Jiang, C., Babar, J., Ciardo, G., Miner, A. S. and Smith, B. 2017. Variable reordering in binary decision diagrams. In *26th International Workshop on Logic and Synthesis*, 1–8.

Johnson, S. G. 2020. The nlopt nonlinear-optimization package.

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press

Kimmig, A., Van den Broeck, G. and De Raedt, L. 2011. An algebraic prolog for reasoning about possible worlds. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. Vol. 1. AAAI Press, 209–214.

Koller, D. and Friedman, N. 2009. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA.

Kraft, D. 1994. Algorithm 733: Tomp–Fortran modules for optimal control calculations. *20,* 3 (Sept.), 262–281.

Latour, A. L. D., Babaki, B., Dries, A., Kimmig, A., Van den Broeck, G. and Nijssen, S. 2017. Combining stochastic constraint optimization and probabilistic programming. In *Principles and Practice of Constraint Programming*, J. C. Beck, Ed. Springer International Publishing, Cham, 495–511.

Lombardi, M. and Milano, M. 2010. Allocation and scheduling of conditional task graphs. *Artificial Intelligence 174,* 7, 500–529.

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R. and Scopatz, A. 2017. Sympy: Symbolic computing in python. *PeerJ Computer Science 3*, e103.

Orsini, F., Frasconi, P., and De Raedt, L. 2017. kProbLog: An algebraic prolog for machine learning. *Machine Learning 106,* 12, 1933–1969.

Riguzzi, F. 2018. *Foundations of Probabilistic Logic Programming*. River Publishers, Gistrup, Denmark.

Riguzzi, F. and Swift, T. 2010. Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In *Technical Communications of the 26th International Conference on Logic Programming (ICLP 2010*. LIPIcs, vol. 7. Schloss Dagstuhl, 162–171.

Rossi, R. A. and Ahmed, N. K. 2015. The network data repository with interactive graph analytics and visualization. In *AAAI*.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming*, L. Sterling, Ed. MIT Press, 715–729.

Sato, T. and Kameya, Y. 2001. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research 15*, 391–454.

Somenzi, F. 2015. *CUDD: CU Decision Diagram Package Release 3.0.0*. University of Colorado.

Svanberg, K. 2002. A class of globally convergent optimization methods based on conservative convex separable approximations. *SIAM Journal on Optimization*, 555–573.

Van den Broeck, G., Thon, I., van Otterlo, M. and De Raedt, L. 2010. DTProbLog: A decision-theoretic probabilistic Prolog. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, M. Fox and D. Poole, Eds. AAAI Press, 1217–1222.

Vennekens, J., Verbaeten, S. and Bruynooghe, M. 2004. Logic programs with annotated disjunctions. In *20th International Conference on Logic Programming (ICLP 2004)*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3131. Springer, 431–445.

Walsh, T. 2002. Stochastic constraint programming. *Proceedings of the 15th European Conference on Artificial Intelligence 1*, 111–115.

Wielemaker, J., Schrijvers, T., Triska, M. and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1–2, 67–96.

https://doi.org/10.1017/S1471068421000260 Published online by Cambridge University Press